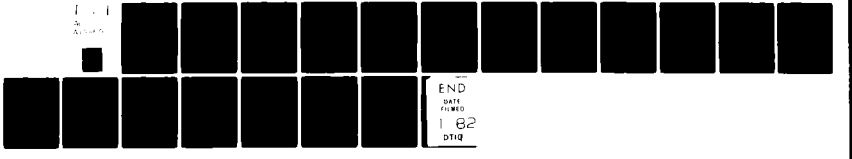
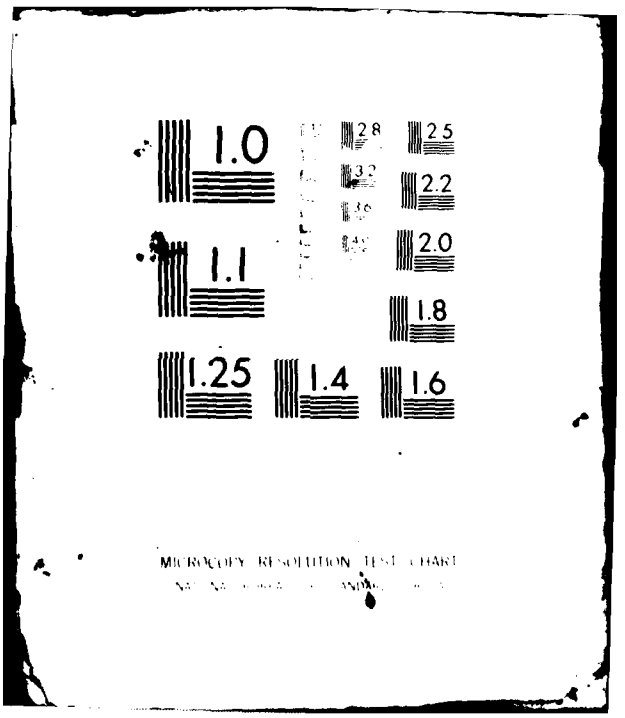
AD-A108 591 MASSACHUSETTS UNIV AMHERST DEPT OF COMPUTER AND INF--ETC F/G 9/2
DEVELOPMENT OF A PROGRAM TESTING SYSTEM.(U)
MAY 81 L A CLARKE AFOSR-77-3287

UNCLASSIFIED AFOSR-TR-81-0803 NL



END
DATE FILMED
1 82
DTIC


1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
MICROCOPY RESOLUTION TEST CHART

Final Scientific Report
for
Development of a Program Testing System

Lori A. Clarke, principal investigator

Grant Number: AFOSR-77-3287
Grant Period: June 1, 1977 - May 30, 1981
Grant Award: $147,348

I. Introduction

Programmers have historically used testing to convince themselves and others that their software works. Although testing has limitations, as do other reliability methods, it has one major advantage: it is the only method in which the actual behavior of the software can be observed. Thus, errors or oversights in the supporting environment, including the available validation tools, translators, operating system, and hardware, may be detected. Moreover, the actual performance can be evaluated for efficiency and usefulness.

We have been using the ATTEST system [CLA78a, CLA78b] as a vehicle to design, implement, and experiment with various aspects of symbolic execution and software testing. ATTEST is an experimental symbolic execution system that attempts to generate test data to satisfy a variety of testing criteria. It is composed of three major components: path selection, symbolic execution, and test data generation. The path selection component is concerned with selecting program paths that satisfy user selected testing criteria. Usually this involves choosing paths that contain untested segments of code. The symbolic execution component then analyzes each selected path. During symbolic execution, the computations on a path are represented as algebraic expressions, the domain for the input values is defined by a set of constraints, and error detection is done. The test data generation component checks the domain constraints for consistency and, if consistent, generates

test data that would drive execution down the selected path.

ATTEST is designed so that it can either augment user-selected test data or independently generate sets of test data. Although ATTEST will not guarantee correctness of a program, it does surpass current testing techniques in that it offers a systematic method of selecting test data to achieve a determined measure of program coverage, and it provides symbolic representations of a path's computation and domain that can be utilized by a variety of testing strategies.

During the grant period, we addressed several issues relating to the development of a testing system. In the sections that follow, an overview of our investigation in each of these issues is provided.

II. Path Selection

A well accepted criterion for testing programs is to execute all branches at least once. Although this criterion does not guarantee detection of all errors, it does guarantee a minimal level of program coverage. Manually creating test data to satisfy this criterion can be very tedious and, at times, very difficult. Experiments have shown that without automated assistance, this level of testing is rarely achieved [STU73]. Furthermore, the sections of code that are the most tedious to test, typically code for checking the validity of parameters and input data, are usually the most susceptible to this approach, thus freeing the programmer to concentrate on more

rigorous testing strategies for the more abstruse code segments.

Previously attempted solutions to the problem of generating a set of paths to execute all branches have been unsuccessful or inefficient. Methods considering only program structure [PAI75, McC76] have failed because of the large number of nonexecutable paths that they generate. Some attempts had been made to detect incompatible branch conditions [KRA73, OST77], however, the complexity of incorporating this information into a static path selection algorithm is NP complete [GAB76]. The SMOTL system [BIC79] statically describes the complete set of paths for executing all branches, but then symbolically evaluates each path to eliminate the nonexecutable ones; this method works, but is extremely inefficient.

We developed an efficient, dynamic method for selecting paths. Instead of describing all the paths before symbolic execution is initiated, this method selects paths during the symbolic execution process. This allows information obtained about a portion of a path to be utilized when selecting the rest of the path. In addition, this method maintains a path history of the program so that it can "learn" from past experience. When a conditional statement is encountered on a path, the current path status and path history are used to determine which branch from this conditional statement should be selected next. The branch that appears to have the highest potential for achieving additional program coverage is selected.

This path selection method is not just applicable to the all branches testing criterion, but is applicable to a range of criteria based on program coverage. Our program coverage method of path selection requires that the path selection criteria be defined by the ATTEST user by specifying, among other things, a relative weight to be given for branch coverage, for statement coverage, and for exercising loop boundary conditions.

In an experiment using the program coverage method [WOO80], the ATTEST system performed efficiently and achieved or approached the specified testing criteria for all programs that were attempted. For example, when a high relative weight was selected for the all branches testing criterion, at least ninety percent of the branches were exercised in all the programs that were analyzed. In addition, only the minimum number of paths was usually required to satisfy the selected testing criterion.

Although the above method has had promising results when testing programs, more comprehensive testing may sometimes be desired. Thus, we also developed and investigated an approach that approximates testing all the paths in a procedure [MAR79]. The problem with an all paths testing criterion is that loops within a procedure may result in an infinite number of paths. Thus, we devised an algorithm for selecting a subset of the paths through a loop that we believe would increase the likelihood of detecting loop boundary errors. Our observations about the best such subset of paths to approximate all paths testing agrees

closely with the subset described in [HOW79]. Except when selecting paths in a loop, this algorithm for path selection employs a depth first search algorithm and uses many of the same techniques that proved successful in the coverage method described above.

III. Array Element Determination

During symbolic execution it is difficult to determine the value of an array index that depends on input values. Although the occurrence of such an indeterminate array index can be represented symbolically for display to the user, such an occurrence usually precludes any further error detection analysis, consistency determination of the path domain, or test data generation. In our experience, this problem has been the major drawback to symbolic execution and has severely limited the types of programs that can be analyzed. Solutions to finding suitable indexes have been proposed [RAM76], but they are not always applicable and are extremely inefficient, often requiring the system to backup to a previous statement on the path.

We have devised a method for handling indeterminate array indexes [MOO79] that seems superior to previously attempted solutions to this problem. This method represents all indeterminate array indexes symbolically, as well as symbolically representing all information that subsequently depends on these indexes. These representations, although too complicated to be meaningful to the user, are internal representations that allow analysis to continue. Using

these representations, the inequality solver in the test data generation component determines a legal range of values for each indeterminate index. It then determines if an index in this range can be found so that the index and its corresponding array element value result in an executable path. If such an index is found, symbolic execution continues. At some later point during symbolic execution the selected array index may no longer be appropriate; the path is nonexecutable unless other indexes satisfying the current set of path constraints can be found. Our array method employs an efficient algorithm for determining the indexes that must be modified and for selecting alternative values if any exist. Moreover, the system does not have to backup; it must only modify the symbolic representations of the effected indexes to correspond to their new values.

We are cautiously optimistic about this algorithm. It takes into account the necessary relationships between array indexes as well as the relationship between each selected index and the value of its corresponding array element. In the worst case, the algorithm degrades to enumeration over all possible array indexes, but this appears to be a rare occurrence. We have manually evaluated this algorithm for several programs and in all cases the algorithm quickly selected appropriate indexes or determined that the paths were nonexecutable.

IV. Test Environment

We completed an initial design for an ATTEST Interface Description language (AID) [WIN78]. In this design we addressed two questions: what capabilities are needed in a symbolic execution session; and how can these requests be easily communicated. The first question led to a command language that tries to capture the testing environment. The second question led to considerable care being taken in designing a natural, uniform, and yet concise syntax for the language.

It is our belief that program testing should be considered throughout software development and not just as an afterthought to the implementation phase. Therefore, the AID language supports a facility for describing a test harness that is useful during the design and implementation phases of software development [OGD78]. Usually the driver program and program stubs in a test harness are simple, static procedures that limit realistic program testing. AID supports a facility for symbolic and dynamic descriptions of these procedures. This capability should help simulate the actual environment more closely and thus allow more realistic testing during software development. Although test data generation is usually not possible during the design phase, symbolic execution resulting in a symbolic representation of the design can be done.

To provide support during the design phase, the AID commands and the programming language statements (in our case FORTRAN) can be intermixed. Initially a procedure can be described predominately by AID commands. As the design proceeds, additional AID commands may be added to refine the design. Gradually the AID commands may be replaced by FORTRAN code. Thus, AID facilitates program design by stepwise refinement.

A study of the use of AID during both the design and testing phases was done [WIN79]. This study revealed several problems with the initial design, the most notable being the lack of a facility to associate AID commands with only the paths related to a particular path selection criterion. For the most part, however, AID was easy to use and provided the desired interface. Although only the most rudimentary features of AID have been implemented in ATTEST, the MUST project [TAY79] is building a symbolic execution system that provides an interface based on AID.

In a related effort, we evaluated three techniques for handling procedure calls in a testing environment. The first and most straightforward technique involves symbolically executing a path in the called procedure at the time the procedure is invoked. This resembles normal execution and requires that all the called procedures be available for symbolic execution. The second technique, called procedure substitution, saves the results from symbolically executing a procedure so that when this procedure is subsequently invoked by another procedure, the

saved results can be 'substituted' for the call. This technique is supportive of bottom-up integration and testing methods in which low level procedures are developed and tested before higher level procedures. An efficient implementation of this technique poses many problems. As our investigation showed [WOO80], it is frequently the case that it is more efficient to symbolically execute a procedure than to substitute the results from a previous execution. Fortunately, an accurate prediction can usually be made about which technique would be more efficient to use for a particular procedure call. The third technique requires that the user, employing AID commands, describe the desired effect of an invoked procedure. This technique supports top down software development where high level procedures are implemented and tested before lower level procedures. AID provides the user with a wide range of capabilities for describing the effects of a procedure call. The descriptions usually provided by a user, however, are simpler to manage than the descriptions automatically provided by procedure substitution, since the user only describes the pertinent testing information. Thus, this third technique can be efficiently implemented.

V. Format Analysis

Since automatic testing systems are concerned with generating legal input data, it is important to examine the corresponding format specifications that exist in languages like FORTRAN and PL/1. Although symbolic execution systems

have been developed for these two languages, none of these systems have taken format specifications into account.

We developed an algorithm to detect data list-format list correspondence during compilation [ABR79]. This algorithm is efficient in that it retains much of the original structure (e.g., iteration counts and nesting) of the initial format specification. A study of 250 I/O statements found that 94% of the data-list, format-list pairs could be completely analyzed at compile time by our algorithm.

Compiler optimization is another area where this algorithm is being successfully applied. Knuth's early study of FORTRAN programs [KNU71] found that twenty-five percent of the execution time was spent in I/O editing, so it is not suprising that this algorithm has been implemented and is now being commercially employed [MCA80] as an optimization technique.

VI. Employing Specifications in the Testing Process

A major drawback of most program testing methods is that program specifications are ignored; test data selection is based solely on the information derived from the implementation. Such methods are unlikely to detect errors that arise when an implementation neglects aspects of the problem, whereas utilizing an understanding of the specification may direct attention to such errors. Recently, several attempts have been made to employ sources of information over and above the implementation in

selecting test data. Goodenough and Gerhart [GOO76] have argued that the specification and the implementation are both valuable sources of information that must be used by testing methods. Thus far, only a few methods [GEL78, WEY80] have been developed to exploit formal specifications, even though such specifications are becoming more readily available as their value in the development of reliable software is recognized.

We have developed a method [RIC78b, 79, 81], called partition analysis, that assists in program testing and program verification by incorporating information from both a formal specification and an implementation for a procedure. The partition analysis method employs symbolic execution techniques to partition the set of input data into procedure subdomains, so that the elements of each subdomain are treated uniformly by the specification and processed uniformly by the implementation. By forming these subdomains, the procedure domain is divided into more manageable units, as is the task of demonstrating program reliability. Information related to each procedure subdomain is used to guide in the selection of test data that reveals errors in the implementation or provides confidence in its correctness. This information is also used to verify consistency between the specification and the implementation. Moreover, the test data selection process, called partition analysis testing, and the verification process, called partition analysis verification, are used to enhance each other; the execution of some elements in the

subdomain may assist in verification, while the verification process may direct the selection of test data.

Our investigation of partition analysis led to some related research endeavors. To create a finite number of subdomains, loop analysis techniques [CHE79] were evaluated and expanded [CLA81b, 81c, 81d]. Work on determining test data for subdomains led to substantial improvements in the Domain Testing Strategy [HAS80] originally proposed by White and Cohen [WHI80]. Finally, in evaluating appropriate specification and design languages for use with partition analysis, recommendations for representing data abstraction and modularity [CLA80] have been proposed as well as more general recommendations for environments supporting software development activities [CLA81a].

VII. Implementation Status

A portion of our time was spent in implementing and evaluating many of the test data generation and symbolic execution features we were investigating. During the grant period our major implementation efforts included the following:

- Three methods of path selection [MAR78, WOO80].
- An efficient system for simplifying constraints [RIC78a].
- An improved interface to the inequality solver that recognizes redundancies and dominance relationships [DIL81].
- Some of the features in the AID language [WIN78].

ATTEST is predominately written in FORTRAN 4 and 5. It is designed to be an experimental system. Storage and execution time considerations have always been given a lower priority than flexibility and adaptability considerations. Originally ATTEST was implemented on the CDC Cyber but was successfully moved to our recently acquired department research facility. ATTEST now runs on the VAX under VMS.

VIII. References

[ABR79] P. Abrahams and L.A. Clarke, "Compile-Time Analysis of Data List-Format List Correspondences," IEEE Transactions on Software Engineering, 5,6, (November 1979), pp.612-617.

[BIC79] J. Bicevskis, J. Borzovs, U. Straujums, A. Zarins, and E. Miller, Jr., "SMOTL - A System to Construct Samples for Data Processing Program Debugging," IEEE Transactions on Software Engineering, SE-5, 1, (January 1979), pp.60-66.

[CHE79] T.E. Cheatham, G.H. Holloway, and J.A. Townley, "Symbolic Evaluation and the Analysis of Programs," IEEE Transactions on Software Engineering, SE-5, 4, July 1979, 402-417.

[CLA78a] L.A. Clarke, "Automatic Test Data Selection Techniques," Infotech State of the Art Report on Software Testing, 2, (September 1978), pp.43-64.

[CLA78b] L.A. Clarke, "Testing: Achievements and Frustrations," IEEE Second International Computer Software and Applications Conference, (November 1978), pp.310-314.

[CLA80] L.A. Clarke, J.C. Wileden, and A.L. Wolf, "Nesting in Ada Programs is for the Birds," Proceedings of the Ada Programming Language Symposium, Boston, Massachusetts, (December 1980), (SIGPLAN Notices, 15,11, (November 1980)), pp.139-145.

[CLA81a] L.A. Clarke, R.M. Graham, and J.C. Wileden, "Thoughts on the Design Phase of an Integrated Software Development Environment," Fourteenth Hawaii International Conference on System Science, (January 1981), pp.11-17.

[CLA81b] L.A. Clarke and D.J. Richardson, "Symbolic Evaluation Methods for Program Analysis," Program Flow Analysis: Theory and Application, editors S. Muchnick and N. Jones, publisher Prentice Hall, (1981).

[CLA81c] L.A. Clarke and D.J. Richardson, "Symbolic Evaluation Methods -- Implementations and Applications," to appear in a North-Holland Publication.

[CLA81d] L.A. Clarke and D.J. Richardson, "Symbolic Evaluation Methods," University of Massachusetts COINS Technical Report, TR81-8, (June 1981), (submitted for publication).

[DIL81] L.K. Dillon, "Constraint Management in the ATTEST System," University of Massachusetts COINS Technical Report, TR81-9, (May 1981).

[GAB76] H.R. Gabow, S.N. Maheshwari, and L.J. Osterweil, "On Two Problems in the Generation of Program Test Paths," IEEE Transactions on Software Engineering, (September 1976).

[GEL78] A. Geller, "Test Data as an Aid to Proving Program Correctness," CACM, 21,5, (May 1978), pp.368-375.

[GOO76] J.B. Goodenough and S.L. Gerhart, "Toward a Theory of Test Data Selection," IEEE Transactions on Software Engineering, SE-1,2, (September 1976), pp.156-173.

[HAS80] J. Hassell, L.A. Clarke, and D.J. Richardson, "A Close Look at Domain Testing," University of Massachusetts COINS Technical Report, TR80-16, (December 1980).

[HOW79] W.E. Howden, "Functional Testing and Design Abstraction," University of Victoria Technical Report, DM-180, IR, (May 1979).

[KNU71] D.C. Knuth, "An Empirical Study of FORTRAN Programs," Software -- Practice and Experience, 1, (1971), pp.105-133.

[KRA73] K.W. Krause, R.W. Smith, and M.A. Goodwin, "Optimal Software Testing Through Automated Network Analysis," Rec. 1973, IEEE Symposium of Software Reliability, (1973).

[MCA80] Massachusetts Computer Associates, Inc., "Proposal for the Evaluation of the FORTRAN Output Expediter," Wakefield, Massachusetts.

[MAR79] L. Marshall, "ATTEST Path Selection Capabilities," University of Massachusetts COINS Technical Note, TN/CS/00043, (December 1979).

[McC76] T.J. McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering, (December 1976).

[MOO79] D.H. Moore, "Arrays Within the Context of Symbolic Execution," University of Massachusetts COINS Technical Note, TN/CS/0042, (August 1979).

[OGD78] N.R. Ogden and L.A. Clarke, "Top-Down Testing with Symbolic Execution," Digest, Workshop on Software Testing and Test Documentation, Ft. Lauderdale, Florida, (December 1978), pp.191-196.

[OST77] L.J. Osterweil, "Data Flow Analysis in Detection of Uninitialized Variables and Editing of Impossible Pairs," TRW Technical Report, (January 1977).

[PAI75] M.R. Paige, "Program Graphs, An Algebra, and Their Implication for Programming," IEEE Transactions on Software Engineering, (September 1975).

[RAM76] C.V. Ramamoorthy, S.F. Ho, and W.T. Chen, "On the Automated Generation of Program Test Data," IEEE Transactions on Software Engineering, SE-2, 4, (December 1976), pp.293-300.

[RIC78a] D.J. Richardson, L.A. Clarke, and D.L. Bennett, "SYMPLR, SYmbolic Multivariate Polynomial Linearization and Reduction," University of Massachusetts COINS Technical Report, TR-78-16, (July 1978).

[RIC78b] D.J. Richardson, "Theoretical Considerations in Testing Programs by Demonstrating Consistency with Specification," Digest, Workshop on Software Testing and Test Documentation (December 1978), pp.19-56.

[RIC79] D.J. Richardson, "Program Testing by Demonstrating Consistency with Specifications," University of Massachusetts COINS Technical Report, TR-79-02, (February 1979).

[RIC81] D.J. Richardson and L.A. Clarke, "A Partition Analysis Method to Increase Program Reliability," Fifth International Conference on Software Engineering, (March 1981), pp.244-253.

[STU73] L.G. Stucki, "Automatic Generation of Self-Metric Software," Rec. 1973, IEEE Symposium Computer Software Reliability, (April 1973), pp.94-100.

[TAY79] R.N. Taylor, R.L. Merilatt, and L.J. Osterweil, "Integrated Testing and Verification System for Research Flight Sotware," Boeing Computer Services Company, (July 31, 1979).

[WEY80] E.J. Weyuker and T.J. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains," IEEE Transactions on Software Engineering, 6,3, (May 1980), pp.236-246.

[WHI80] L.J. White and E.I. Cohen, "A Domain Strategy for Computer Program Testing," IEEE Transactions on Software Engineering, 6,3, (May 1980), pp.247-257.

[WIN78] D. Winters, N.R. Ogden, and L.A. Clarke, "A Definition of AID -- The ATTEST Interface Description Language," University of Massachusetts COINS Technical Report, TR78-15, (December 1978).

[WIN79] D. Winters and L.A. Clarke, "A Testing Experiment Using AID," University of Massachusetts COINS Technical Note, TN/CS/00044, (August 1979).

[WOO80] J.L. Woods, "Path Selection for Symbolic Execution Systems," Ph. D. Dissertation, Computer and Information Science, University of Massachusetts, September 1980.

IX. Papers and Reports During the Grant Period

L.A. Clarke and J. Woods, "Program Testing Using Symbolic Execution," Proceedings of the Software Specification and Testing Technology Conference, Washington, D.C., (April 1978), pp.124-144.

D.J. Richardson, L.A. Clarke, and D.L. Bennet, "SYMPLR -- Symbolic Multivariate Polynomial Linearization and Reduction," University of Massachusetts COINS Technical Report, TR78-16, (July 1978).

L.A. Clarke, "Automatic Test Data Selection Techniques," Infotech State of the Art Report, Software Testing, 2, Publisher Infotech International Limited, (September 1978), pp.43-65.

L.A. Clarke, "Testing: Achievements and Frustrations," IEEE Second Internatinal Computer Software and Applicatons Conference, Chicago, (November 1978), pp.310-314.

N.R. Ogden and L.A. Clarke, "Top-Down Testing with Symbolic Execution," Digest, Workshop on Software Testing and Test Documentation, Ft. Lauderdale, Florida, (December 1978), pp.191-196.

D.J. Richardson, "Theoretical Considerations in Testing Programs by Demonstrating Consistency with Specifications," Digest, Workshop on Software Testing and Test Documentation, Ft. Lauderdale, Florida, (December 1978), pp.19-56.

D. Winters, N.R. Ogden, and L.A. Clarke, "A Definition of AID -- the ATTEST Interface Description Language," University of Massachusetts COINS Technical Report, TR78-15, (December 1978).

D.H. Moore, "Arrays Within the Context of Symbolic Execution," University of Massachusetts COINS Technical Note, TN/CS/00042, (August 1979).

L. Marshall, "ATTEST Path Selection Capabilities," University of Massachusetts COINS Technical Note, TN/CS/00043, (September 1979).

P. Abrahams and L.A. Clarke, "Compile-Time Analysis of Data List-Format List Correspondences," IEEE Transactions on Software Engineering, 5,6, (November 1979), pp.612-617.

L.A. Clarke, J.C. Wileden, and A.L. Wolf, "Nesting in Ada Programs is for the Birds," Proceedings of the Ada Programming Language Symposium, Boston, Massachusetts, (December 1980), (SIGPLAN Notices, 15,11, (November 1980), pp.139-145).

J. Hassell, L.A. Clarke, and D.J. Richardson, "A Close Look at Domain Testing," University of Massachusetts COINS Technical Report, TR80-16, (December 1980).

L.A. Clarke, R.M. Graham, and J.C. Wileden, "Thoughts on the Design Phase of an Integrated Software Development Environment," Fourteenth Hawaii International Conference on System Science, (January 1981), pp.11-17.

D.J. Richardson and L.A. Clarke, "A Partition Analysis Method to Increase Program Reliability," Fifth International Conference on Software Engineering, (March 1981), pp.244-253.

L.A. Clarke and D.J. Richardson, "Symbolic Evaluation Methods for Program Analysis," Program Flow Analysis: Theory and Applicationl, editors S. Muchnick and N. Jones, publisher Prentice Hall, (1981).

L.A. Clarke and D.J. Richardson, "Symbolic Evaluation Methods -- Implementations and Applications," to appear in a North-Holland Publication.

L.K. Dillon, "Constraint Management in the ATTEST System," University of Massachusetts COINS Technical Report, TR81-9, (May 1981).

L.A. Clarke and D.J. Richardson, "Symbolic Evaluation Methods," University of Massachusetts COINS Technical Report, TR81-8, (June 1981), (submitted for publication).